TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky, informatiky a mezioborových studií

Studijní program: B2612 – Elektrotechnika a informatika
Studijní obor: Informatika a logistika

Využití lexikální analýzy při stavbě modelů

The use of lexical analysis in the construction of models

Bakalářská práce

Autor: **Ondřej Nedvídek**
Vedoucí práce: doc. Ing. Dalibor Frydrych, Ph.D.
Konzultant: Ing. Jan Lisal

V Liberci 31.5.2009

Prohlášení

Byl(a) jsem seznámen(a) s tím, že na mou bakalářskou práci se plně vztahuje zákon č. 121/2000 o právu autorském, zejména § 60 (školní dílo).

Beru na vědomí, že TUL má právo na uzavření licenční smlouvy o užití mé bakalářské práce a prohlašuji, že **s o u h l a s í m** s případným užitím mé bakalářské práce (prodej, zapůjčení apod.).

Jsem si vědom(a) toho, že užít své bakalářské práce či poskytnout licenci k jejímu využití mohu jen se souhlasem TUL, která má právo ode mne požadovat přiměřený příspěvek na úhradu nákladů, vynaložených univerzitou na vytvoření díla (až do jejich skutečné výše).

Bakalářskou práci jsem vypracoval(a) samostatně s použitím uvedené literatury a na základě konzultací s vedoucím bakalářské práce a konzultantem.

Datum

Podpis

Poděkování

Rád bych poděkoval doc. Ing. Daliboru Frydrychovi, Ph.D za pomoc při vypracování této bakalářské práce a za jeho trpělivost při konzultacích.

Abstrakt

Práce se zabývá počítačovou aplikací, která řeší matematické rovnice zadávané programu v textové podobě. Rovnice jako jsou jednoduché součty, přes složitější rovnice, až po složité rovnice s N neznámými, jsou předány lexikálnímu analyzátoru, který určí jednotlivé výrazy v textu. Tyto výrazy jsou předány pro zpracování syntaktickému analyzátoru, který si vytvoří strukturu pro následné vyhodnocení rovnice.

Pro řešení aplikace je použito návrhových vzorů a objektového programovacího jazyka Java. Hlavními znaky aplikace jsou její robustnost a modifikovatelnost a tím i univerzálnost použití. Právě univerzálnost a modifikovatelnost by měla zajišťovat její využití v širokém spektru matematických modelů.

Klíčová slova: OOP, Java, Lexikální analýza, Syntaktická analýza, Návrhové vzory, Interpreter

Abstract

This work deals with computer application, which solves mathematical equations input into the program in text form. Equations, such as simple additions as weel as complex equations, up to the complex equations with N variables, are transfered to scanner, which identifies individual terms in the text. Further, these terms are transfered to parser for processing. Parser creates a structure for subsequent evaluation of the equation.

To address this application, design patterns and object programming language Java are used. The main features of the application are its robustness and modifiability and thus the versatility of utilization. The very versatility and modifiability should provide its application in a wide range of mathematical models.

Keywords: OOP, Java, Scanner, Parser, Design Patterns, Interpreter

Obsah

1. Úvod

Vývoj počítačových programů prošel a stále prochází intenzívním vývojem. Na počátku byla tvorba počítačových programů spíše doménou osamělých programátorů. Pracovní výkon programátora závisel na jeho vlastních znalostech a intuici. Stále širší používání informačních technologií ve všech oborech lidské činnosti a stále rostoucí požadavky na schopnosti počítačových programů si vynutily spolupráci více pracovníků na jednom programu, a proto vznikají firmy zabývající se výhradně tvorbou programů (software house). Do procesu vývoje programu se tak dostávají další prvky; sdílení informací mezi pracovníky, zajištění kontinuity vývoje při fluktuaci pracovníků, zajištění termínů uvádění nových programů na trh, zajištění kvality prodávaných programů. Pro úspěšné přežití firmy na trhu je nutné mít možnost sledovat produktivitu práce jednotlivých pracovníků, a pokud možno ji zvyšovat. Tlak na zvyšování produktivity práce s sebou přinesl nové postupy používané při vývoji programů.

V mnoha případech dochází k opakovanému řešení stále stejných problémů (evidence osob, evidence bankovních účtů, vedení účetnictví), byť s drobnými nuancemi. Vývoj programů se stává průmyslovou výrobou, s jejími standardy, normami a kontrolou kvality. Standardní řešení se označují jako „návrhové vzory“. První návrhové vzory byly publikovány v knize Design Patterns: Elements of Reusable Object-Oriented Software [1]. V této knize bylo publikováno 23 základních návrhových vzorů. V současné době jsou návrhové vzory rozšířeny do většiny oblastí vývoje programů (databáze, run-time, atd.).

Standardizace při programování matematických modelů však dosud není příliš rozšířená. Je to patrně dáno širokou variabilitou řešených problémů a samou podstatou matematiky, jakožto velice abstraktní vědy. Tato práce se pokouší ukázat cestu, jak by bylo možné zefektivnit vývoj a použití matematických modelů.

V mnoha matematických modelech jsou používány různé funkce, které jsou používány k výpočtům jistých hodnot. Sama „funkce“ je abstraktní pojem. V modelu pak musí být implementována jedna nebo více konkrétních funkcí. Kterou konkrétní funkci chce uživatel použít, specifikuje pomocí parametru. V případě, že by uživatel chtěl použít konkrétní funkci, která není v modelu implementována, musel by kontaktovat autora daného matematického modelu, který by model o tuto funkci rozšířil. Úspěšnost tohoto procesu je velice malá (zda nám autor vyhoví závisí na jeho

dobré vůli), a když tak časově a finančně (v případě, že jde o komerční program) náročná.

Je možné přesunout konkretizování použité funkce blíže k uživateli matematického modelu? Řešením je použít „lexikální analyzátor“. Uživatel matematického modelu zadá konkrétní funkci jako textový řetězec a lexikální analyzátor zajistí výpočet její hodnoty pro dané proměnné.

Tento problém byl již řešen mnoha autory. Všechna publikovaná řešení však byla realizována pro konkrétní názvy použitých proměnných. Tím je použitelnost těchto lexikálních analyzátorů výrazně snížena. Až na úroveň jednorázové řešení, kdy pro každý model s jinými proměnnými bude nutné znovu implementovat, buď celý lexikální analyzátor a nebo alespoň jeho část.

Při tvorbě lexikálního analyzátoru publikovaného v této práci byly zohledněny tyto požadavky:

- možnost začlenění do uživatelského programu BEZ nutnosti modifikovat zdrojový kód lexikálního analyzátoru
- možnost začlenění do uživatelského programu BEZ nutnosti VÝRAZNÉ modifikace zdrojového kódu matematického modelu
- přizpůsobení pro konkrétní funkčnost, případně rozšíření o nové funkce
- rychlost srovnatelná s nativním řešením

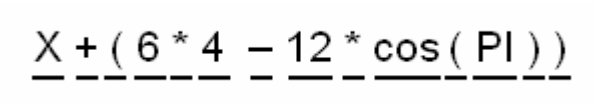
První tři body charakterizují úspěšně vyřešenou znovu použitelnost. Lexikální analyzátor je implementován pouze jednou (autorem lexikálního analyzátoru). Jeho konkrétní chování, např. na které proměnné má reagovat, se určuje při jeho inicializaci (kterou určuje programátor matematického modelu). Tvar konkrétní funkce pak specifikuje uživatel matematického modelu.

2. Teorie interpreteru

Gramatika použitá v kódu programu, na základě níž je vstupní řetězec interpretován, závisí na klasické matematické posloupnosti vykonávání operací. To se pak projevuje i na výsledné struktuře rozložení zadaného textu funkce. Použita byla stromová struktura, do které jsou postupně umístěny všechny jednotlivé části kódu, a úrovňově pak odpovídá hierarchickému vykonávání dílčích operací.

2.1 Rozčlenění textu

Jak bylo zmíněno již v úvodu, text je důležité před vytvořením jeho stromové struktury nějak rozdělit na výrazy.



Obr.1 – Rozčlenění textu

Na obrázku jsou podtrženy jednotlivé výrazy, podle kterých se struktura vytvářeného stromu řídí. Následně pak již není problém vytvořit stromovou strukturu, reprezentující daný matematický výraz.

2.2 Gramatika

Ohledně gramatiky bylo nejprve nutné určit požadavky, které by měla daná gramatika splňovat. Námi daná gramatika tedy umožňuje použití těchto výrazů:

- Binární operace – sčítání, odčítání, násobení, dělení, mocnění a priorita vykonávání operací
- Závorky – práce se závorkami
- Funkce – jednoparametrové funkce
- Proměnné
- Konstanty

Procházení textu se provádí stejně jako u klasického čtení, tedy zleva doprava. Proto jsou jednotlivé znaky načítány postupně, jak je čteme, nikoli však, jak bychom s nimi počítali. Počítání ovlivňuje jak priorita operandů, tak závorky. Proto bylo nutné vytvořit určitou gramatiku, podle které se budou jednotlivé prvky textu rozřazovat do stromu. Gramatika pro určení pozice ve stromové struktuře pak vypadá takto.

```
term := sum
sum := product | sum ‘+‘ sum | sum ‘-‘ sum
product := value | product ‘*‘ product | product ‘/‘ product
value := element | ‘-‘ element | ‘+‘ element
element := constant | variable | function | ‘(‘ term ‘)‘
constant := number
variable := word
function := word‘(‘ term ‘)‘
```

Z dané gramatiky je vidět, že se text interpretuje tak, že považován za nejabstraktnější část, tedy výraz (term) a pak postupuje strukturou, jak je dáno v určené gramatice (zjednodušeně řečeno směrem dolů).

V našem případě je určen první výraz, tedy písmeno „x“. Ten se „propadá“ v hierarchii gramatiky dolů, kde se určí, že se jedná o proměnnou, tedy nějaký znakový řetězec (slovo). Dalším výrazem je znaménko „+“, to je určeno při postupu gramatikou zpět, tedy směrem nahoru. Výsledkem určení je, že se jedná o součtový znak, tedy znak se dvěma operandy. Jeho prvním operandem je tedy proměnná „x“ a druhým operandem je výraz v závorkách. To, že se jedná o celý výraz, určují závorky, které se nacházejí na úrovni element. Tento výraz je pak rozložen podobným způsobem. Lépe to je vidět na přímém rozkladu do stromové struktury.

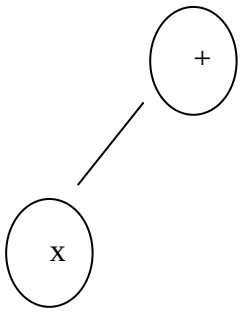
2.3 Rozklad do stromové struktury

V kapitole 2.2 jsme nastínili, jaký je postup určování prvků pomocí gramatiky. V této kapitole bude ukázáno, jak jsou jednotlivě gramaticky určené výrazy řazeny do stromové struktury.

Zjednodušeně řečeno jsou na nejvyšší pozici ty členy, které jsou v gramatice nejvýše postaveny. Pokud se v textu nachází gramaticky stejně postavených výrazů

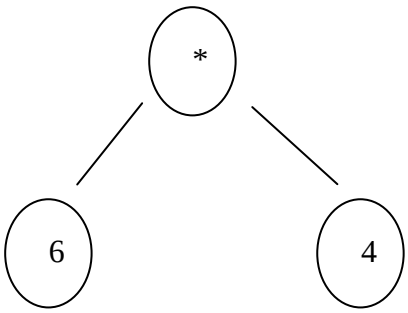
více, jejich pozici určuje ještě pozice v interpretovaném textu. Výrazy více nalevo jsou výše než ty napravo. Takto zjednodušený pohled je ale maličko zavádějící a při použití závorek je již nutno myslet i na další zákonitosti.

Proto zde na obrázcích bude názorně uvedeno, jak takový rozklad funguje. Pro ukázku bude použita stejná funkce, jaká je uvedena v kapitole 2.1. V kapitole 2.2 je uveden již začátek rozkladu, kdy jsme si určili dva výrazy z textu a to proměnnou „x“ a operátor „+“. Proměnná „x“ byla navíc určena také jako první operand operátoru „+“.



Obr. 2 – Rozklad na strom 1.

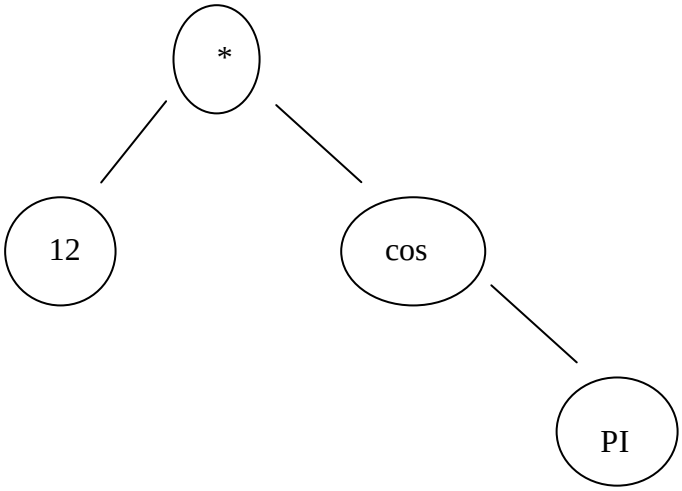
Jako druhý operand je určen termín v závorce. Abychom mohli tento operand připojit, je nutné rozložit text v této závorce a až následně je tato větev připojena ke stromové struktuře. Jelikož je postup stejný, jako v předchozím případě, první výraz, který je určen, je číslice „6“. Po průchodem gramatikou je určena jako konstanta a při návratem gramatikou zpět nahoru je nalezen další výraz. Tímto výrazem je operátor násobení „*“. Konstanta „6“ je tedy určena jako jeho první operand. Druhý operand je hledán opět průchodem gramatikou směrem dolů, ale důležité je, že od úrovně, kde byl určen operand násobení. Druhým operandem je pak určena konstanta „4“. Jelikož za konstantou již není žádný znak nacházející se gramaticky pod úrovní násobení, ba naopak je zde znaménko „-“, které je gramaticky výše, je rozklad pro násobení hotov a stává se zároveň prvním operandem pro operátor „-“. Rozklad pro operátor „*“ tedy vypadá následovně.



Obr. 3 – Rozklad na strom 2.

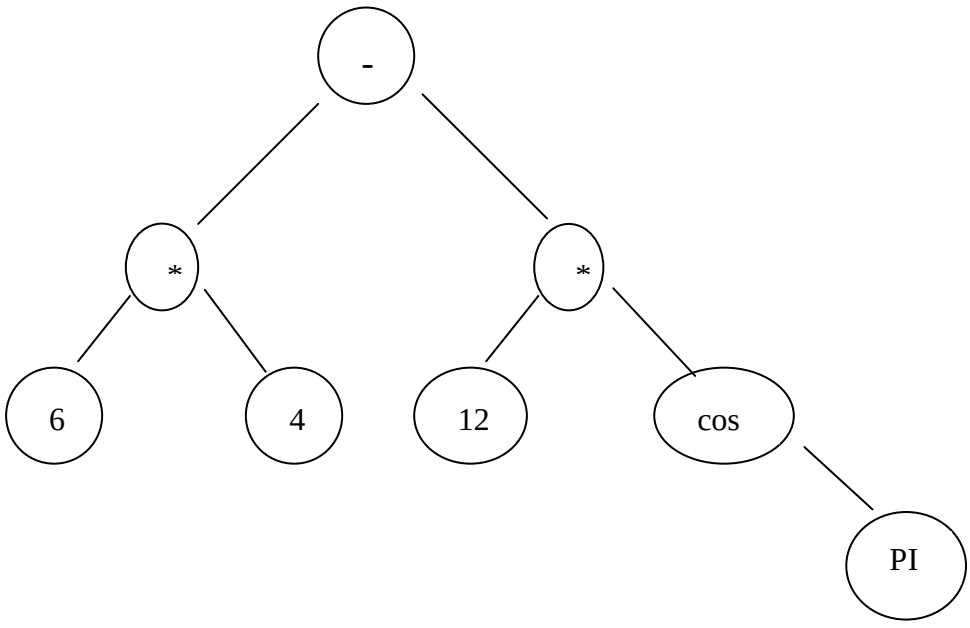
Máme-li první operand pro operátor „-“, je nutné určit jeho druhý operand. Ten je určen podobně jako první. Kdy je nejdříve nalezeno číslo „12“, určené jako konstanta. Při návratu gramatikou zpět je hledán další znak a je nalezen opět operátor násobení, operátor úrovně product. Konstanta 12 je určena jako jeho první operand a je hledán operand druhý. Druhý operand, řetězec „cos“, je hledaný opět směrem dolů v gramatice. Po průchodem gramatikou jazyka je tento řetězec určen jako funkce.

Každá funkce má pak parametr v závorkách. Tento parametr je určen jako výraz, nebo-li term. Term se nachází na samém vrcholu gramatiky. Při jeho určování je s ním zacházeno jako s rozkladem celé funkce. Po průchodu gramatikou je určeno, že se jedná o konstantu „PI“. Když je určen parametr funkce, je navrácen funkci a tím je určený i druhý operand pro druhé násobení v zadané funkci. Celá větev pro toto násobení pak vypadá takto.



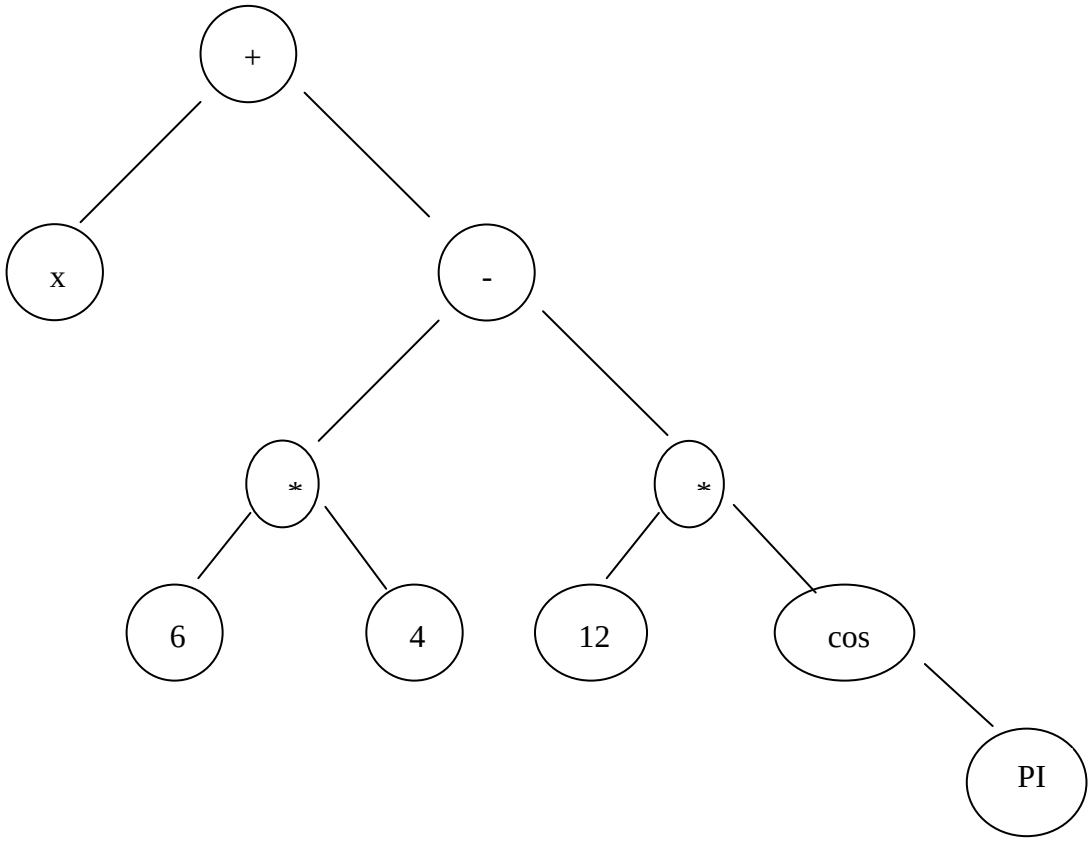
Obr. 4 – Rozklad na strom 3.

Teď tedy máme určeny oba operandy operátoru „-“ a můžeme si složit celou strukturu pro tento operátor. Ta vypadá následovně.



Obr. 5 – Rozklad na strom 4.

Tímto je zároveň určen i celý výraz v závorce. Jak bylo uvedeno výše, tak výraz v závorce je druhým operandem pro operátor součtu „+“. Výraz v závorce pak připojíme přes jeho kořen, jímž je operátor „-“. Tímto je tedy určena celá stromová struktura zadané funkce a vypadá jednoduše takto.



Obr. 6 – Finální stromová struktura

3. Programátorské nástroje

Pro realizaci programu byl použit programovací jazyk Java a technologie návrhových vzorů, zejména pak návrhového vzoru *Interpreter*.

3.1 Java

Jazyk Java byl zvolen jako vývojový nástroj hned z několika důvodů. Java je čistě objektově orientovaný jazyk. Pokud nepočítáme osm základních datových typu, pak vše ostatní v tomto jazyce jsou objekty. Tato vlastnost sama o sobě tvoří z Javy nástroj pro tvorbu velmi robustních a lehko modifikovatelných aplikací.

Další obrovskou výhodou tohoto jazyka je přenositelnost kódu na různých platformách, což z tohoto jazyka dělá jeden z nejuniverzálnějších jazyků vůbec. Java je jazykem interpretovaným, její kód je tedy při kompilaci převeden na ByteCode (mezikód). Podmínkou pro běh programů v tomto jazyce je tedy funkčnost Java Virtual Machine (JVM) na dané platformě, protože pouze pomocí JVM jsme schopni ByteCode přeložit a spustit.

Velkou výhodou Javy je také její rozšířenost a to zejména v posledních několika letech. Toto s sebou nese velké množství studijních materiálů, ať už zdrojových kódů, informací v diskuzích, samotných knižních publikacích a v neposlední řadě také oficiální dokumentace společnosti Sun [2]. Jinými slovy je jazyk Java jedním z nejrozšířenějších a nejvíc podporovaných jazyků vůbec. Tomuto napomáhá také fakt, že tento programovací jazyk je open source a jeho používání je tedy zdarma.

3.2 Návrhové vzory

Návrhové vzory jsou v podstatě metodiky, jak obecně postupovat v určitých opakujících se případech při vytváření programu. Návrhové vzory (angl. Design patterns), pak je nutno pouze specifikovat pro určitý případ použití. Tím je myšleno, že návrhové vzory neřeší, jeden určitý problém, ale spíše postup, jak podobné problémy řešit.

Návrhové vzory jsou navíc maximálně odladěny tak, aby byla jejich implementace přehledná a zároveň robustní a snadno rozšiřitelná. To dělá z aplikací řešených návrhovými vzory stabilní programy, které ani po případném dalším zásahu do

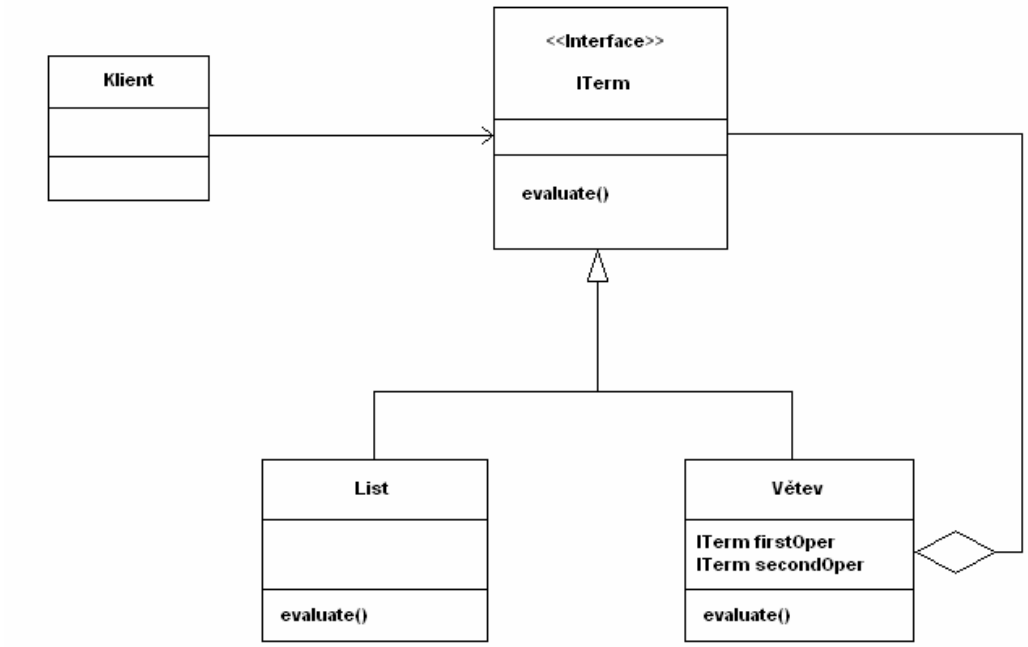
kódu, například z důvodu doplnění dalších funkcí apod., nezpůsobí žádné komplikace. To však platí za předpokladu, že se při zásahu budeme řídit opět návrhovými vzory.

3.2.1 Návrhový vzor *Composite*

Návrhový vzor *Composite* popisuje, jak vytvořit strukturu jednoduchých a složených objektů (z toho také vychází název vzoru composite = složený). Složené objekty jsou pak právě ty, co se skládají z jednoduchých. Navíc nám říká, jak vše uspořádat, aby se právě k oběma typům přistupovalo stejnou cestou.

Všechny tyto prvky jsou řazeny do stromové struktury, která má jeden hlavní prvek, tedy kořen. Z kořene pak vystupují jeho potomci, tedy jednoduché objekty, zvané též listy, nebo objekty složené, označované jako větve. Větve se dále dělí na další větve a listy a tak to postupuje až k nejnižším listům.

Návrhový vzor *Composite* je pak řešením, jak docílit toho, aby bylo přistupováno k těmto větvím a listům stejným způsobem.



Obr. 7 – Návrhový vzor *Composite*

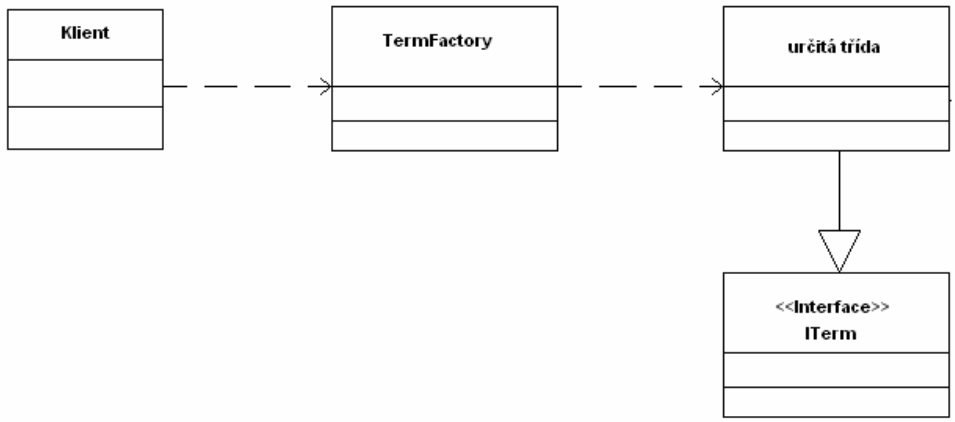
Jak je vidět na obrázku 7, návrhový vzor *Composite* funguje tak, že jak pro list, tak pro větev má nadefinováno jedno rozhraní, v našem případě označeno jako ITerm. Přes toto rozhraní pak k oběma objektům přistupuje nezávisle na tom, který z nich to je.

Je-li v rozhraní nějaká metoda, tuto metodu umí i jednoduché a složené objekty. Zatím co u jednoduchých objektů je metoda přímo vykonána, u složených je metoda předána k vykonání dalším objektům, které se nacházejí ve stromové struktuře pod nimi. Takto postupuje až do té chvíle, dokud těmi objekty nejsou opět objekty jednoduché.

3.2.2 Návrhový vzor *Factory method*

Tento návrhový vzor zjednodušuje práci s vytvářením instancí jednotlivých tříd. Hlavní zjednodušení pak spočívá v jednoduchém přidávání dalších tříd a tím rozšiřování rodiny tříd, jejichž instance je návrhový vzor *Factory method* schopen vracet.

Funkčnost pak závisí na použití jednotného rozhraní ITerm pro všechny třídy vytvářených objektů. Vracené objekty jsou pak typu rozhraní ITerm a klient tedy v podstatě neví, jaký objekt mu byl navrácen. Klient je odkázán pouze na třídu TermFactory, které může předat pouze parametry, se kterými bude třída TermFactory pracovat. Jinak je zcela oddělen od principu výběru třídy, která bude objekt vytvářet. Třída TermFactory tedy navrací objekt „určité třídy“, která implementuje rozhraní ITerm. Následně pak klientovi navrácen objekt vybraný třídou TermFactory.



Obr. 8 – Návrhový vzor *Factory*

3.2.3 Návrhový vzor *Interpreter*

Návrhový vzor *Interpreter* je jednou z možností, jak se vypořádat s interpretací nějakého jednoduchého jazyka. Abychom mohli tohoto vzoru použít na vlastní

interpretaci, je nutné určit dané řídící znaky jazyka. Termín řídící znak byl použit záměrně, protože jimi nemusíme myslet pouze slova nebo čísla. Jazyk, který interpretujeme, totiž nemusí mít textovou podobu, ale může jím být v podstatě cokoliv. Důležité však je pro každý typ z těchto řídících znaků vytvořit jeho vlastní třídu. Vyskytne-li se pak daný řídící znak v kódu jazyka, je pak z jeho třídy vytvořen objekt. Všechny vytvořené objekty jsou řazeny do stromové struktury a to podle gramatiky interpretovaného jazyka.

Gramatika je další nedílnou součástí návrhového vzoru *Interpreteru*, je tedy důležité si na ní dát velice záležet. Čím složitější jazyk interpretujeme, tím je gramatika složitější a je zde nebezpečí vytvoření těžko odhalitelných chyb.

Stromová struktura, je realizována pomocí návrhového vzoru *Composite*. K jednotlivým částem vytvořeného stromu je pak přistupováno (jak již bylo řečeno v kapitole 3.2.1.) přes rozhraní, které implementují všechny vytvořené objekty.

4. Implementace

Vlastní aplikace je tedy napsána v jazyce Java, přesněji ve vývojovém prostředí NetBeans IDE 6.1 společnosti Sun.

4.1 Popis funkce

Jak bylo popsáno v kapitole 2., program se zabývá řešením funkcí s libovolným počtem nezávisle proměnných. Funkce jsou zadávány programu jako textový řetězec, který je v první řadě zbaven všech bílých znaků a až následně je předán dál k lexikální a syntaktické analýze. Tyto dvě analýzy probíhají v podstatě současně. Postup je takový, že při lexikální analýze je zjištěn výraz, který je ve stejný moment i vyhodnocen syntaktickou analýzou a je navrácen objekt třídy, která výrazu odpovídá a v závislosti na gramatice je pak umístěn do stromové struktury. Tímto způsobem program postupuje až do posledního znaku řetězce a pak navrátí stromovou strukturu zadané funkce.

Při vyhodnocování funkce je pak stromová struktura neměnná a výsledek se odvíjí od hodnot proměnných a konstant. Hodnoty proměnných se pak můžou měnit, ale stromová struktura zůstává stále stejná. Stejná zůstává až do chvíle, dokud nezadáme v textovém řetězci jinou funkci. Ta musí být opět předána lexikálnímu a syntaktickému analyzátoru a převedena do nové stromové struktury.

4.2. Rozhraní ITerm

Toto rozhraní je základem všech tříd, z kterých jsou vytvářeny objekty pro stromovou strukturu, tedy tříd, které odpovídají řídícím znakům (viz kapitola 3.2.3) zadávané funkce. Lze tedy přes něj přistupovat ke všem částem výsledného stromu (viz kapitola 3.2.1).

Jedinou metodu, kterou implementuje toto rozhraní je pak metoda evaluate(), která vrací hodnotu. Hodnota, kterou vrátí, pak závisí na volaném prvku ve stromové struktuře.

4.3 Elements

V této třídě se nachází dva základní stavební kameny a to konstanty a proměnné. Pouze objekty těchto dvou tříd určují základní hodnotu a od nich se následně odvíjí hodnoty všech ostatních objektů ve stromové struktuře. Je tedy jasné, že se ve stromech nacházejí na pozicích listů a jsou zároveň jedinými prvky, které se na této pozici vyskytují.

4.3.1 Třída Constant

Tato třída implementující rozhraní ITerm je nadefinována tak, že vytvářený objekt si převede a uloží si vlastní hodnotu, předanou konstruktoru řetězcem, do vnitřní finální proměnné typu double. K převodu používá metody parseDouble(), což zajistí převod na double i konstantu zapsanou v exponenciálním tvaru (více viz kapitola 4.7.1, metoda interpConstant()).

Při volání metody evaluate() pak pouze objekty třídy Constant vrací svou hodnotu.

```
public class Constant implements ITerm{

    final double value;

    public Constant(String s){
      this.value = Double.parseDouble(s.toString());
    }

    public double evaluate() {
        return value;
    }
}
```

4.3.2 Třída Variable

S třídou Variable je to maličko složitější než s třídou Constant. Tato třída si při vytváření instance uloží vnitřní proměnnou term typu ITerm. Tuto proměnnou získá

z mapy variableMap, která je vytvořena v každém objektu třídy Interpreter. V této mapě se nachází názvy proměnných a k nim určené jejich třídy. Proměnná takto řeší universálnost zadávání hodnot proměnných při vyhodnocování funkce. Nezáleží pak na tom z jakého zdroje je proměnná odebírána a je pouze důležité aby objekt třídy tohoto zdroje implementovala rozhraní ITerm a vracel tedy svojí hodnotu přes metodu evaluate().

Tímto způsobem jsou řešeny i základní konstanty PI a E, které mají nadefinovány vlastní třídy implementující rozhraní ITerm a metoda evaluate() pak pouze vrací jejich hodnotu, která je určena pomocí třídy Math. Po předání variableMap objektu třídy Interpreter je pak vždy naplněna těmito základními konstantami.

```
public class Variable implements ITerm {
    private ITerm term;

    public Variable(String name, Map<String, ITerm> variableMap){
        if (variableMap.containsKey(name)) {
            term =  variableMap.get(name);
        }else throw new IllegalStateException("Zadana promenna nebyla
deklarovana!!!");
    }

    public double evaluate(){
        return term.evaluate();
    }
}
```

4.4 Balíčky operators a functions

Operátory a funkce jsou zbylými stavebními prvky stromu, pokud jsou tedy konstanty a proměnné listy stromu, pak je jejich pozice určena jako jeho větve. A to jak větve z kterých vycházejí další dva prvky, tedy operators a nebo prvek pouze jeden, tedy functions.

4.4.1 Balíčky operatorsSum a operatorsProduct

Tyto dva balíčky shromažďují funkčně v podstatě stejné třídy. Tyto třídy dokonce rozšiřují stejnou abstraktní třídu ABinaryOperation implementující též

rozhraní ITerm. V této abstraktní třídě se nachází pouze dva vnitřní parametry typu ITerm a to firstOper s secondOper. Tyto dva parametry představují operandy pro binární operátory, které právě všechny třídy rozšiřující rozhraní ABinaryOperation představují.

Důvodem pro rozdělení operátorů do těchto dvou balíčku je jejich priorita při vyhodnocování. Není-li určeno jinak (pomocí závorek), tak je přednostně vyhodnocován objekt z balíčku operatorsProduct, kde se nachází třídy pro násobení, dělení atd., následně pak objekty balíčku operatorsSum se třídami pro sčítání a odčítání.

Použití jednotlivých tříd, tedy jejich znalost interpreterem, je pak určena jejich přidáním do sumMap respektive productMap. Znalost dalších operací lze tedy jednoduše vyřešit přidáním třídy rozšiřující ABinaryOperation a přidáním odkazu do sumMap nebo productMap v závislosti na požadované prioritě.

```
public class Plus extends ABinaryOperation{

    public Plus(ITerm firstOper, ITerm secondOper){
        this.firstOper = firstOper;
        this.secondOper = secondOper;
    }

    public double evaluate(){
        return firstOper.evaluate() + secondOper.evaluate();
    }
}
```

4.4.2 Balíček functions

V tomto balíčku se nacházejí všechny funkce jedné proměnné. Abstraktní třída AFunction, podobně jako ABinaryOperation implementuje rozhraní ITerm. Na rozdíl od ní však má pouze jeden vlastní parametr param typu ITerm.

Podobně jako u operators je pak znalost funkcí určena též mapou a to functionMap. Znalost dalších funkcí je řešena podobně jako u operators, přidáním třídy rozšiřující AFunction a odkazu do functionMap.

```
public class Sin extends AFunction {
    public Sin(ITerm param){
```

```
        this.param = param;
    }

    public double evaluate(){
       return Math.sin(param.evaluate());
    }
}
```

4.4.3 Třídy SumMap, ProductMap a FunctionMap

Jak již bylo naznačeno výše, tak tyto mapy uchovávají informaci o tom, jaké binární operátory a funkce interpreter zná. Tedy co v textu rozezná a s čím dokáže pracovat. Tyto mapy jsou typu HashMap<String, String> a na základě nich jsou objekty vytvářeny pomocí tovární metody, které jsou předávány. Metoda pro naplnění mapy je vidět níže.

```
public static Map fillProductMap(){
        Map<String,String> productMap = new HashMap<String, String>();
        productMap.put("*","operatorsProduct.Multiplication");
        productMap.put("/","operatorsProduct.Divide");
        productMap.put("^","operatorsProduct.Power");
        return productMap;
    }
```

Jak je vidět, mapa na základě klíče vrací pouze cestu ke třídě v textové podobě. Tato cesta je pak předávána faktory metodě ve třídě TermFactory.

4.5 Třída TermFactory

Koncepce tovární metody (*Factory method)* byla zvolena tak, aby její použití bylo co nejuniversálnější. Tovární metoda zobecňuje proces tvorby nových instanci a umožňuje tak jednoduché rozšiřování rodiny tříd, jejichž instance daná tovární metoda vytváří. Správně implementovaná tovární metoda pak realizuje proces rozšíření rodiny tříd, bez zásahu do stávajícího kódu aplikace i kódu samotné tovární metody. Tímto postupem se výrazně snižuje možnost nechtěného zavlečení chyby do zdrojových kódů. Stačí pouze vložit cestu v textové podobě do příslušné mapy (viz výše) a tím je

zaručeno, že tovární metoda bude umět vytvořit objekt dané třídy. Za předpokladu, že třída existuje a je správně navržena.

Třída TermFactory obsahuje dvě metody, které se liší pouze v počtu předávaných parametrů. První metoda slouží k vytváření objektů se dvěma parametry, v našem případě se jedná o objekty tříd binárních operátoru. Předávanými parametry jsou pak jejich operandy, tedy firstOperand a secondOperand. Druhá metoda, je pro vytváření jednoparamterových funkcí. Lze ji použít i pro vytváření konstant nebo proměnných, nicméně toto řešení by bylo zbytečně složité a postrádalo by smysl.

Obě tyto metody pak fungují stejně. Je jim předána cesta k třídě a určitý počet parametrů. Cesta je v našem případě uložená do proměnné className. Třída, na základě cesty skládající se ze jména balíčku a jména třídy, získá požadovanou třídu. Následně za pomocí knihovny java.lang je vytvořen objekt daného konstruktoru na základě počtu a typu parametru. Za pomocí tohoto konstruktoru je pak vytvořena sama instance objektu dané třídy a ten je následně předán.

```
public static ITerm createTerm(final String className, ITerm param){
        ITerm term = null;
        try {
            Class clazz = Class.forName(className);
            java.lang.reflect.Constructor con =
clazz.getConstructor(ITerm.class);
            term = (ITerm) con.newInstance(param);
        }catch (InstantiationException ex) {
            ex.printStackTrace();
        } catch (IllegalAccessException ex) {
            ex.printStackTrace();
        } catch (IllegalArgumentException ex) {
            ex.printStackTrace();
        } catch (InvocationTargetException ex) {
            ex.printStackTrace();
        } catch (NoSuchMethodException ex) {
            ex.printStackTrace();
        } catch (ClassNotFoundException ex) {
            ex.printStackTrace();
        }
        return term;
    }
```

4.6 Třída Crate

Crate je anglický výraz pro přepravku. Toto slovo taky vystihuje hlavní podstatu této třídy, respektive objektu této třídy. Přepravka slouží k přepravování více věcí. V našem případě pak bude přepravovat důležitou informaci a to samotnou textovou podobu zadané funkce (proměnna text). Dalšími informacemi jsou pozice, na které se v textu nacházíme (pos) a zároveň znak na této pozici (actualChar).

Pro navracení znaku na aktuální, předcházející pozici slouží metody getChar() a previousChar() a pro pozici následující voláme metodu nextChar(). Další metoda je zde ještě setActualChar(), která nastaví actualChar podle pozice, kterou ji předáme.

```
public class Crate {
    public final String text;
    public int pos = 0;
    private char actualChar;

    public Crate(String text){
        this.text=text;
        actualChar = text.charAt(pos);
    }

    public char nextChar(){
        pos = pos + 1;
        actualChar = text.charAt(pos);
        return actualChar;
    }

    public void setActualChar(int pos){
        this.pos = pos;
        actualChar = text.charAt(pos);
    }

    public char getChar(){
        return actualChar;
    }

    public char previousChar(){
        return text.charAt(pos-1);
    }
```

4.7 Třída Interpreter

Srdce celého programu se nachází v této třídě. Tady je vykonávána lexikální i syntaktická analýza. Tedy objektu této třídy je předána textová podoba funkce. A za pomocí jeho metody interpTerm() je vrácena stromová struktura. Neboli text je rozdělen na řídící znaky, ty jsou následně určeny a jsou vytvořeny jejich objekty. Do stromové struktury jsou tyto objekty vloženy podle gramatiky interpretovaného jazyka. Stromová struktura je předána jako proměnná typu ITerm.

Realizovaná gramatika, je již popsána v kapitole 2.. Pro větší názornost jsem upravil některé části a označil je tučně.

```
term := interpSum
sum := interpProduct | sum ‘+‘ interpSum | sum ‘-‘ interpSum
product := interpValue | product ‘*‘ interpProduct |
           product ‘/‘ interpProduct
value := interpElement | ‘-‘ element | ‘+‘ element
element := interpConstant | interpVariable | interpFunction |
‘(‘ interpTerm ‘)‘
constant := number
variable := word
function := word‘(‘ interpTerm ‘)‘
```

Na takto formulovanou gramatiku se budu odkazovat v dalším popisu, kde bude jasnější, co tedy jednotlivé části znamenají.

4.7.1 Interpreter a jeho metody

Konstruktor

Konstruktoru je předán text určený k lexikální a syntaktické analýze. V konstruktoru je text optimalizován a je předán jako parametr pro instanci třídy přepravka. Přepravka text uchová a slouží, jako jediný zdroj informací, kde se v rozkládaném textu nacházíme, pro všechny metody interpreteru. Ať se tedy na přepravku odkáže jakákoliv část, vždy ji je vrácený, právě ten znak, který je momentálně řešený.

Dalším předávaným parametrem, je mapa proměnných (variableMap). V této mapě se nacházejí všechny proměnné, které bude v danou chvíli interpreter umět použít a zároveň (viz. kapitola 4.3.2) i základní konstanty (PI, E).

Nakonec jsou v konstruktoru naplněny ještě ostatní mapy, tedy productMap, sumMap a functionMap, jimiž jsou dány funkce a binární operace interpreteru.

```
public Interpreter(String text, Map<String, ITerm> variableMap){
        text = TextEdit.VymazMezery(text);
        crate = new Crate("("+text+") ");
        productMap = ProductMap.fillProductMap();
        sumMap = SumMap.fillProductMap();
        functionMap = FunctionMap.fillProductMap();
        this.variableMap = VariableMap.fillVariableMap(variableMap);
    }
```

Metoda interpTerm()

Toto je jediná metoda interpreteru, kterou můžeme volat zvenčí, je tedy jako jediná public. Právě touto metodou odstartujeme interpretaci zadané funkce. Jak je vidět z kódu programu a zároveň z gramatiky (kapitola 4.7), jediné na co se metoda interpTerm() odkazuje a zároveň jediné co vrací je výsledek metody interpSum(). Vrácený výsledek pak může být tím konečným, nebo právě pouze interpretací nějaké části interpretované funkce.

```
public ITerm interpTerm(){
        return interpSum();
    }
```

Metoda interpSum()

V této metodě je v první řadě určena proměnná sum typu ITerm, ta je určena jako výsledek metody interpProduct(), jinými slovy je předána dál. Teorie je pak taková, že pokud je text validně zadán, pak všechny binární operace mají dva operandy. Tedy je vždy nejdříve určen první operand, než bude zjišťováno, zda vůbec se ve funkci ten daný operator vyskytuje (v případě interpSum(), jde o operatory v sumMap).

Potom co je v proměnné sum nějaký výraz typu ITerm (konstanta, proměnná, součin, atd..), je zjištěno, jestli na aktuální pozici přepravky se nachází znak z mapy sumMap. Pokud ano, je předána informace z mapy tovární metodě a ta vytvoří objektovou reprezentaci tohoto operátoru. Prvním operandem operátoru je proměnná sum a druhým operandem bude výraz stejné úrovně jako je sum, nebo úrovně gramaticky nižší, tedy všechny se nacházející v gramatice pod sum. Tím je zajištěna priorita vykonávání operací.

```
private ITerm interpSum(){
        ITerm sum = interpProduct();
        //System.out.println("InterpSum: "+crate.getChar());
        if(sumMap.containsKey(String.valueOf(crate.getChar()))){
            crate.nextChar();
            sum =
TermFactory.createTerm(sumMap.get(String.valueOf(crate.previousChar())
), sum, interpSum());
        }
        return sum;
    }
```

Na konci pak pouze metoda vrátí vzniklou část stromové interpretace. Tato interpretace je pak vrácena buďto metodě interpTerm() a nebo se stejně tak může stát operandem opět metody interpSum().

Metoda interpProduct()

Tato metoda funguje téměř identicky s metodou interpSum, jediným avšak gramaticky podstatným rozdílem je, že první operand je brán jako výsledek metody interpValue() a druhý jako výsledek metody interpProduct(). Tedy oba operandy jsou brány gramaticky o úroveň níž. Jiná je samozřejmě také mapa, z které je vyhodnocováno zda-li jde o operátor úrovně product. Stejně, jako metoda interpSum(), vrátí vzniklou část stromové interpretace. Je tedy patrné, že v tomto případě může být součin právě jedním z operandů pro operátor úrovně sum, ale i pro úroveň product.

Metoda interpValue()

Metoda interpValue() slouží k určení hodnoty daného výrazu úrovně element. Pokud je tedy znak actualChar „-“ nebo „+“ a předchozí znak je nějaký operátor nebo levá závorka (toto řeší metoda isItItForValue()), vyhodnotí se znak jako znaménko určující zápornost, nebo kladnost elementu. V případě kladného znaménka je tento znak v řetězci pouze přeskočen a je volána metoda interpElement(). V případě záporného znaménka je případ vyhodnocen, jako součin konstanty „-1“ a výrazu nacházejícího se bezprostředně za ním, vráceného metodou interpElement().

Pokud není actualChar ani jedním ze znamének, je automaticky volána metoda interpElement().

```
private ITerm interpValue(){
        ITerm value = null;
        if (crate.getChar()=='+' && isItItForValue()){
            crate.nextChar();
            value = interpElement();
        }else if(crate.getChar()=='-' && isItItForValue()){
            crate.nextChar();
            value = new Multiplication(new Constant("-
1"),interpElement());
        }else{
            value = interpElement();
        }
        return value;
    }
```

Metoda interpElement()

V této metodě se rozhoduje na základě hodnoty v actualChar, do jaké ze tří skupin patří. Zda se jedná o konstantu, funkci nebo proměnnou, případně výraz v závorkách. Jestliže na základě znaku je určeno, že se nejedná ani o jednu z těchto skupin, je vracena prázdná proměnná element = null typu ITerm.

Pokud je actualChar číslice, je další řešení předáno metodě interpConstant(). Je-li actualChar písmeno, je volána metoda interpVariableOrFunction() a pokud se jedná o levou závorku, volá se metoda interpTerm(). Pokud jde o závorku, to co je mezi

nimi, je řešeno jako samostatný výraz. Po jeho vrácení je již pouze vyhodnoceno, zdali je závorka uzavřena. Je-li uzavřena, metoda interpElement() vrací daný výraz.

Stejně tak vrací výraz, pokud je vyhodnocena konstanta nebo proměnná s funkcí.

```
private ITerm interpElement(){
        ITerm element = null;

        if(Character.isDigit(crate.getChar())) {
            element = interpConstant();
        }else if(Character.isLetter(crate.getChar())) {
            element = interpVariableOrFunction();
        }else if(crate.getChar()=='(') {
            crate.nextChar();
            element = interpTerm();
            if(crate.getChar()!= ')'){
                except();
            }else{
                crate.nextChar();
            }
        }else
            except();
        return element;
    }
```

Metoda interpConstant()

V této metodě je realizováno rozpoznávání konstant. Konstanta je načítána znak po znaku od prvního výskytu číslice. Z důvodu, že číslice mohou být načítány ve tvaru s desetinou tečkou (například „0.01“), nebo ve tvaru s exponentem (například „10e-3“), je nutné správně ošetřit i výskyt dalších znaků. Ošetřeny jsou i znaky „e“, „.“, „+“ a „-“. Pokud máme načtený řetězec po sobě správně jdoucích znaků a čísel, je tento řetězec odeslán konstruktoru třídy Constant k dalšímu zpracování. O zbytek se pak postará metoda parseDouble() třídy Double.

```
private ITerm interpConstant(){
        int endPos = crate.pos; //konecna pozice pritomnosti cisla v
retezci
        StringBuffer s = new StringBuffer();
```

```
        while(crate.text.charAt(endPos)=='.' |
Character.isDigit(crate.text.charAt(endPos)) |
crate.text.charAt(endPos)=='e' ){
            if(crate.text.charAt(endPos)=='e' &&
crate.text.charAt(endPos+1)=='+' | crate.text.charAt(endPos+1)=='-'){
                s = s.append(crate.text.charAt(endPos));
                endPos++;
            }
            s = s.append(crate.text.charAt(endPos));
            endPos++;
        }
        crate.setActualChar(endPos); // nastavi pozici za konstantu
        return new Constant(s.toString());
    }
```

Metoda interpVariableOrFunction()

Tato funkce pracuje podobně jako metoda interpConstant(). Také je zde načítán řetězec znaků a číslic. Aby tato metoda byla zavolána, je zde podmínka, že prvním znakem načítaného řetězce je písmeno. Jinými slovy, jak funkce, tak proměnná jsou určeny, jako náhodně dlouhý řetězec po sobě jdoucích písmen a číslic, začínající písmenem.

Tato metoda ale určuje, zda se jedná o funkci nebo proměnnou. Textově se tyto dva řídící znaky od sebe odlišují ve výskytu levé závorky za názvem (řetězcem znaků a číslic). Na základě závorky tedy určíme, zda se jedná o proměnnou, nebo funkci.

Je-li řetězec proměnná, je volána třída Variable, které je předán řetězec dané proměnné a mapa proměnných variableMap. Pokud se v řetězci vyskytuje levá závorka, jde o funkci a je volána metoda interpFunction() s předaným parametrem jménem funkce.

```
private ITerm interpVariableOrFunction(){
        ITerm variableOrFunction = null;
        int endPos = crate.pos;
        StringBuffer nameBuffer = new StringBuffer();
        while(Character.isLetterOrDigit(crate.text.charAt(endPos++)))
            nameBuffer =
                  nameBuffer.append(crate.text.charAt(endPos-1));
        if(crate.text.charAt(endPos-1)=='('){ // pokud zavorka, tak
```

```
            crate.setActualChar(endPos);
            variableOrFunction = interpFunction(nameBuffer.toString());
            return variableOrFunction;
          }
          crate.setActualChar(endPos-1);
         variableOrFunction =
        new Variable(nameBuffer.toString(),variableMap);
          return variableOrFunction;
  }
```

Metoda interpFunction()

Tato metoda vrací funkci reprezentovanou názvem předaným parametrem. Jedinou podmínkou je výskyt funkce v mapě functionMap. Metoda nejprve zjistí parametr funkce, tak že zavolá metodu interpTerm(). Poté co metoda interpTerm navrátí hodnotu zpět, je ověřeno, zda je funkce zakončena závorkou. Pokud je i tato podmínka splněna, je předána tovární metodě TermFactory.createTerm() cesta k třídě, dané funkce (je-li definována ve functionMap) a její parametr. Metoda pak navrací objekt typu ITerm zpět.

```
private ITerm interpFunction(String className){
        ITerm param = interpTerm();
        if(crate.getChar()!= ')'){
                except();
            }
        crate.nextChar();
        return TermFactory.createTerm(functionMap.get(className),
param);
    }
```

4.8 Testy

Testování bylo rozděleno na dvě základní části. Pří výpočtu funkční hodnoty je pak vždy nejdůležitější přesnost výsledku. Druhým, však neméně důležitým aspektem, je pak rychlost vypočtu dané funkce. V určitých případech je pak důležité najít kompromis mezi rychlostí a přesností výpočtu.

4.8.1 Testování přesnosti výpočtu hodnoty funkce

Testování přesnosti bylo provedeno pro dva typy výpočtu. Oba dva typy byly testovány na jednom výrazu (Rovnice 1.).

$$f = \pi * 3 - \cos(\pi / 2)$$

Rovnice 1.

V prvním případě byla k dosažení výsledku použita sama aplikace. Tedy stanovená rovnice byla podrobena lexikální a syntaktické analýze a byla rozložena do stromové struktury. Následně pak byla stromová interpretace vyhodnocena a byl navrácen výsledek. V druhém případě pak bylo dosaženo výsledku za pomocí výpočtu ve standardním kódu programovacího jazyka Java. Oba výsledky pak byly vyjádřeny v datovém typu double a tím byla dána i maximální přesnost s jakou byly výsledky vypočteny.

Výsledek vyhodnocení interpretované stromové struktury:

- f = 9,42477796076938

Výsledek kódu jazyku Java:

- f = 9.42477796076938

4.8.2 Testování rychlosti výpočtu hodnoty funkce

K testování bylo přistupováno podobně, jako v předchozím případě. Rychlost výpočtu byla testována, jak pro vyhodnocení výpočtu stromového vyjádření funkce, tak pro výpočet v kódu jazyka Java. Test probíhal v počtu $N=200^3$ kroků, jelikož by za jiných podmínek byla doba výpočtu prakticky neměřitelná. Jelikož bylo více kroků, tak bylo nutno rozdělit případ, kdy je rozložen výraz na stromovou strukturu a vyhodnocen na dva další případy. Tedy první případ, kdy je v každém kroku vytvořen strom funkce a zároveň je v tomto kroku i vyhodnocen. Druhý případ, kdy je vytvořen strom funkce pouze jednou a v dalších krocích je pak tento stejný strom pouze vyhodnocován.

Test rychlosti pro všechny tři případy byl proveden pro dvě rovnice. Aby byly zohledněny jednoduché výpočty výrazů bez funkcí (Rovnice 2.) a dále pak i výpočty s větším počtem funkcí (Rovnice 3.). Právě u výpočtů s větším počtem funkcí je

předpokládán menší rozdíl v rychlostech, jelikož bude větší část celkového času vynaložena na samotné vyhodnocování funkcí.

$$f = 6 + 4 / 2 - 3{,}25$$

Rovnice 2.

Výsledky testu pro Rovnici 2.

Výsledek N-krát interpretované a N-vyhodnocené stromové struktury:

- 02 : 28,406 [mm:ss]

Výsledek N-krát interpretované a N-vyhodnocené stromové struktury:

- 00 : 00,375 [mm:ss]

Výsledek nativního kódu:

- 00 : 00,016 [mm:ss]

$$f = \pi * e - (\cos(\pi / e) * \sin(3) + \tan(\pi / 4 + \pi))$$

Rovnice 3.

Výsledky testu pro Rovnici 3.

Výsledek N-krát interpretované a N-vyhodnocené stromové struktury:

- 06 : 20,922 [mm:ss]

Výsledek 1-krát interpretované a N-vyhodnocené stromové struktury:

- 00 : 04,735 [mm:ss]

Výsledek nativního kódu:

- 00 : 03,562 [mm:ss]

5. Závěr

V této práci je popsán vývoj a implementace lexikálního analyzátoru. Práce se skládá ze tří částí.

První část, se zabývá teorii lexikální analýzy matematických vztahů (funkcí). Je zde definována gramatika matematických vztahů, priorita matematických operací, závorek apod..

V druhé části, jsou popsány použité nástroje k implementaci. Pro implementaci byl použit objektový programovací jazyk Java. S úspěchem zde bylo použito standardizovaných řešení, tzv. návrhových vzorů. Díky jejich využití, vyšla vlastní implementace, jako kompaktní, snadno čitelná a dobře dokumentovaná.

Pro zajištěni rychlosti byl lexikální analyzátor rozdělen do dvou samostatných částí. První část se zabývá rozborem uživatelem v textové formě zadané funkce. Pro tuto část, bylo použito standardizované řešení pomocí návrhového vzoru *Interpreter*. Opakovaná lexikální analýza při výpočtu stejně zadané funkce, by znamenala neúměrný nárůst výpočetního času. Proto byl výpočet funkční hodnoty vyčleněn do druhé části. Pro tuto část existuje optimální řešení s použitím standardizovaného návrhového vzoru *Composite*. Časově náročná lexikální analýza se pak provádí pouze jedinkrát a pro opakovaný vypočet funkční hodnoty je použito optimálnější a rychlejší řešení. Tento postup zajišťuje splnění požadavku rychlosti řešení, srovnatelného s nativním řešením.

Pro splnění požadavku znovu použitelnosti a snadného začlenění do uživatelského programu, bylo při realizaci lexikálního analyzátoru využito dynamických datových struktur. Všechny „znalosti“ lexikálního analyzátoru (jaké matematické operace a funkce je schopen realizovat) jsou vytvářeny při inicializaci. Pro inicializace je s úspěchem použit návrhový vzor *Factory method*. Tímto postupem se přidání dalších „znalostí“ do lexikálního analyzátoru redukuje na přidání nové třídy s realizaci dané znalosti a zaregistrování dané třídy v projektu lexikálního analyzátoru. Tento postup zcela eliminuje možnost zavlečeni chyby do stávající implementace.

V celé implementaci je důsledně použito programování oproti rozhraní. Pro začlenění lexikálního analyzátoru do uživatelského programu je nutné, pouze implementovat jediné rozhraní ITerm. Opět se jedná se o zcela minimální zásah, bez možnosti zavlečení chyby do uživatelského programu.

Lexikální analyzátor, jehož implementace je popsána v této práci, splňuje všechny požadavky specifikované v zadání. Tím, že jeho „schopnosti“ se specifikují až při samotné inicializaci, je jeho využití zcela obecné a pokrývá širokou oblast matematických modelů. Je tak zajištěna jeho snadná a efektivní znovu použitelnost.

Seznam použité literatury

[1] GAMMA Erich, HELM Richard, JOHNSON Ralph, VLISSIDES M. John. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional Computing Series, 1994. ISBN 978-0201633610. 416 str.

[2] *Java™ Platform, Standard Edition 6.* [online]. [cit. 2009-05-22]. Oficiální materiály společnosti Sun Microsystems.
URL: <http://java.sun.com/javase/6/docs/api/index.html/>

[3] PECINOVSKÝ Rudolf. *Návrhové vzory.* Computer Press, a.s., 2007. Vydání první. ISBN 978-80-251-1582-4. 527 str.

[4] HEROUT Pavel. *Učebnice jazyka JAVA.* KOPP, 2008. Rozšířené vydání. ISBN 978-80-7232-355-5. 381 str.

[5] HEROUT Pavel. *JAVA bohatství knihoven.* KOPP, 2008. Třetí vydání. ISBN 978-80-7232-368-5. 245 str.

[6] DVOŘÁK Miloš. *Návrhové vzory (design patterns).* [online]. [cit 2009-05-14]. URL: <http://objekty.vse.cz/Objekty/Vzory>.

[7] *Lexikální analýza. Wikipedia.org* [online]. [cit. 2009-05-14]. URL: <http://cs.wikipedia.org/wiki/Lexikální_analýza>.

[8] *Syntaktická analýza.* Wikipedia.org [online]. [cit. 2009-05-14]. URL: <http://cs.wikipedia.org/wiki/Syntaktická_analýza>.

[9] *A look at the Composite design pattern.* JAVAWORLD – Solutions for JAVA developers. [online]. [cit. 2009-05-14]. URL: <http://www.javaworld.com/javaworld/jw-09-2002/jw-0913-designpatterns.html>

[10] *Interpreter pattern.* School of Computer Science. [online]. [cit. 2009-05-14]. URL: <http://www.cs.mcgill.ca/~hv/classes/CS400/01.hchen/doc/interpreter/interpreter.html>